Brad Abrams

Designing Microsoft® .NET Class Libraries

June 2004

Lesson 5: Designing Inheritance Hierarchies

- After successfully completing this lesson, you will be able to:
  - Understand how to build useful inheritance hierarchies
  - Correctly use interfaces or base classes

Dangers of Over-Designing

- The most common library design mistake
- A new developer or program manager is designing the widget feature:
  - Want it to be extensible and feature-rich
  - The reality of shipping hits
    - Design is only partly thought out
    - Long bug tail
    - Forced hard cuts
    - We ship it cumbersome and broken
  - In V.next
    - We now know the extensibility needed
    - But blocked by the broken design that shipped
- Moral
  - Do as little as possible now (but no less) to ensure room for extensibility in the future

Abstract and Base Classes

- Base classes serve as the root of an inheritance hierarchy
- Abstract classes are a special kind of base class that are non-instantiable and may contain members that aren't implemented
- Prefer broad, shallow hierarchies
  - Less than or equal to two additional levels = rough rule!
- Contracts and responsibilities are difficult to maintain and explain in deep complex hierarchies

Abstract and Base Classes

- Consider making base classes not constructible (example: use abstract classes)
  - Make it clear what the class is for
  - Provide a protected constructor for subclasses to call
  - System.Exception should not have had a public constructor

Virtual, Abstract, and Non-Virtual

- **Virtual** members are points of specialization or callbacks in your code

```
public virtual int Length { get {..} }
```

- **Abstract** members are required points of specialization in your code

```
public abstract int Read()
```

- **Non-virtual** members cannot be overridden in derived types

```
public string Remove (int index, int count)
```

Virtual Methods

```
public class TheBase : Object {
    public override string ToString() {
        return "Hello from the Base";
    }
}
```

```
public class Derived : TheBase {
    public override string ToString() {
        return "Hello from Derived";
    }
}
```

Virtual Methods

- What is printed out?

```
Derived d = new Derived();
Console.WriteLine (d.ToString());
```

```
TheBase tb = d;
Console.WriteLine (tb.ToString());
```

```
Object o = tb;
Console.WriteLine (o.ToString());
```

Virtual Methods

- Virtual methods all output "Hello from Derived"—why?
  - Method call virtualizes at run time
  - The static type doesn't matter
- This is the danger and power of virtual methods
  - Danger: Owner of base classes cannot control what subclasses do
  - Power: Base class doesn't have to change as new subclasses are created

Overriding

- Don't change the semantics of member
  - Follow the contract defined on the base class
- Don't require clients to have knowledge of your overriding
- Consider whether you should call the base implementation
  - Choose to call it unless you have good reason not to

Virtual and Non-Virtual

- Use non-virtual members unless you have specifically designed for specialization
  - Have a concrete scenario in mind
  - Write the code!
- Think before you virtualize members
  - Modules that use references to base types must be able to use references to derived types without knowing the difference
    - Must continue to call in the same order and frequency
    - Cannot increase or decrease range of inputs or output
  - See the Liskov Substitution Principle

Abstract Members

- Methods, properties, and events can be abstract
- Use abstract members only where it is absolutely required that subclasses provide a custom implementation
- Only use when the base class cannot have any meaningful default implementation

Abstract, Virtual, and Non-Virtual Members

- Default to making members non-virtual
- Make virtual if it is designed to be specialized by subclasses
- Make abstract if no meaningful default implementation is possible
  - Unless versioning issues prohibit it, in which case throw a NotImplementedException

Interfaces vs. Base Classes

- Choose to use base classes over interfaces
  - Base classes version better in general
    - Allows for adding members
  - Members can be added with a default implementation
    - Avoids incompatibilities common in Microsoft® ActiveX®
  - Interfaces are good for versioning behavior (changing semantics)

Interfaces vs. Base Classes

- Avoid having both base class and interfaces
  - Adds confusion about which to use
    - Component vs. IComponent
  - Little advantage
- Consider using aggregation
- Don't use attributes where a contract is needed

Aggregation

- Some of the needs for multiple inheritance can be solved with aggregation
- Instead of having C derive from A and B, have C derive from A and have member that returns an instance of B

```
// Not supported
public class C : A, B {}
```

```
// Right
public class C : A {
      public B MyB {get {..}}
}
```

Versioning of Interfaces

```
interface IStream // version 1
{
    void Read (byte[] value);
}
```

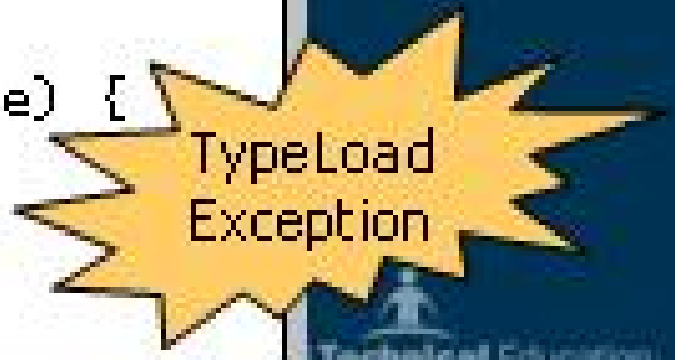
```
class FileStream: IStream {
    public void Read (byte[] value) {
        ...
    }
}
```

Versioning of Interfaces

- Add a Write method in version 2 of the interface
- Version 1 clients are broken

```
interface IStream  // version 2
{
    void Read (byte [] value);
    void Write (byte [] value);
}
```

```
class FileStream: IStream
{
    public void Read (byte [] value) {
        ...
    }
}
```



Versioning of Base Classes

```
public class Stream    // version 1
{
   public virtual void Read (byte[] value) {
      ...
   }
}
```

```
public class FileStream: Stream
{
  public override virtual void Read (byte[] value){
     ...
  }
}
```

Versioning of Base Classes

```
public class Stream   // version 2
{
  public virtual void Read (byte[] value){...}
  public virtual void Write (byte[] value){...}
}
```

```
public class FileStream: Stream
{
  public override virtual void Read (byte[] value){
    ...
  }
}
```

- FileStream continues to work with default implementation for `Write()`

Interface Usage

```
public interface IComparable {
      int CompareTo(object obj);
}
```

- Interfaces are useful!
- The smaller and more focused the interface the better
  - 1–2 members are best
  - But interfaces can be defined in terms of other simpler interfaces
  - Examples: IComparable, IFormattable

Explicit Method Implementations

- Implementing members of an interface "privately"
  - Not a security boundary!
- Only accessible when cast to the interface type
- Hides implementation details
  - Clean public interface
    - IConvertible on Int32, etc.
  - Differentiates implementations
  - Simpler strong typing

Explicit Method Implementations: Clean Public Interface

- The 19 ToXxx methods on IConvertible don't "pollute" the Int32 public view
  - But they are there when cast to IConvertible
  - Solution: Implement them privately

```
public struct Int32 : IConvertible, IComparable {
    public override string ToString () {..}
    int IConvertible.ToInt32 () {..}
    ...
}
```

```
int i = 42;
i.ToString();  // works
i.ToInt32();   // does not compile
((IConvertible) i).ToInt32();  // works
```

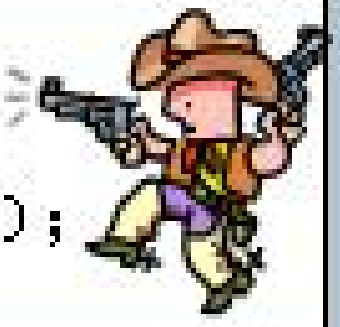
Explicit Method Implementations: Differentiates Implementations

- Interfaces developed by different groups can have the same signature for different meanings
  - `Draw()` a picture and `Draw()` a gun
- Frequently you want to differentiate the implementation
- Explicit method implementations enables this
- Avoids us recommending "unique" names in interfaces

Explicit Method Implementations: Differentiates Implementations

```
interface IGraphics {
   void Draw();
   Brush Brush { get; set; }
   Pen Pen { get; set; }
}
```

```
interface IBandit {
   void Draw();
   void Duel(IBandit opponent);
   void Rob(Bank bank, IBandit[] sidekicks);
}
```



```
class Bandit: IBandit, IGraphics {
   void IBandit.Draw() {...}
   void IGraphics.Draw() {...}
}
```

Exercise: Choose the Right Root

- Define a type that describes a contract for comparing this instance to another

Exercise: Choose the Right Root

- Define a type that describes a contract for all kinds of employees in the database

Lesson 5 Summary

- Use virtual methods carefully
- Choose base classes over interfaces
- Base classes version better than interfaces
- Explicit method implementation can be useful